Interview Question: Binary Search Trees
Write pseudo-code to perform an in-order traversal in a binary search tree without using recursion.

Solution:
This algorithm is implemented as the BST Iterator in P2. Check it out!

Recurrences and Closed Forms
For the following code snippet, find a recurrence for the worst case runtime of the function, and then find a closed form for the recurrence.

Consider the function $f$:

```c
1  f(n) {
2      if (n == 0) {
3          return 1;
4      }
5      return 2 * f(n - 1) + 1;
6  }
```

- Find a recurrence for $f(n)$.

Solution:

$$T(n) = \begin{cases} 
  c_0 & \text{if } n = 0 \\
  T(n-1) + c_1 & \text{otherwise}
\end{cases}$$

- Find a closed form for $f(n)$.

Solution:

Unrolling the recurrence, we get $T(n) = c_1 + c_1 + \cdots + c_1 + c_0 = c_1 n + c_0$. 
Recurrences and Closed Forms

For the following code snippet, find a recurrence for the worst case runtime of the function, and then find a closed form for the recurrence.

Consider the function $g$:

```java
1  g(n) {
2    if (n < 3) {
3        return 1000;
4    }
5    if (g(n/3) > 5) {
6        for (int i = 0; i < n; i++) {
7            System.out.println("Yay!");
8        }
9        return 5 * g(n/3);
10    }
11    else {
12        for (int i = 0; i < n * n; i++) {
13            System.out.println("Yay!");
14        }
15        return 4 * g(n/3);
16    }
17 }
```

• Find a recurrence for $g(n)$.

**Solution:**

Note that the `else` statement will never actually happen in practice. The solution to $g(n)$ is always greater than 5 (in fact, greater than 1000).

$$T(n) = \begin{cases}
  c_0 & \text{if } n = 1 \\
  2T(n/3) + c_1 n & \text{otherwise}
\end{cases}$$

• Find a closed form for $g(n)$.

**Solution:**

Let’s take an inventory of all of the numbers we have to consider.

- Branching Factor / Number of Function Calls from each Recursive Case Function Call $b = 2$
- Reduction in Size of $N$ for each step $r = 3$
- Work in Each Recursive Function Call $w_{rec} = c_1 n$
- Work in Base Case / Final Function Calls $w_{base} = c_0$
- Height of Recursion Tree / Number of Function Calls until Base Case $h = \log_r n = \log_3 n$
- Number of Base Case Calls / Leaf Nodes of Recursive Tree $n_{base} = b^h = 2^{\log_3 n}$

Now, let’s build a summation to evaluate the recursive case.

$$\sum_{i=0}^{h} \left( \frac{b}{r} \right)^i w_{rec} + n_{base} w_{base} = \sum_{i=0}^{\log_3(n)-1} \left( \frac{2}{3} \right)^i c_1 n + 2^{\log_3 n} c_0$$

Let’s simplify this equation. We will take advantage of a few uncommon properties. The sum of the finite geometric series $\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$, $|x| < 1$ and the nifty log property $n^{\log_a x} = x^{\log_a n}$. 
\[
\sum_{i=0}^{\log_3(n) - 1} \left( \frac{2}{3} \right)^i c_1 n + 2^{\log_3 n} c_0 = c_1 n \sum_{i=0}^{\log_3(n) - 1} \left( \frac{2}{3} \right)^i + 2^{\log_3 n} c_0 \\
= c_1 n \left( 1 - \left( \frac{2}{3} \right)^{\log_3(n)} \right) + c_0 n^{\log_3(2)} \\
= c_1 n \left( 1 - \left( \frac{2^{\log_3(n)}}{3^{\log_3(n)}} \right) \right) + c_0 n^{\log_3(2)} \\
= 3c_1 n \left( 1 - \frac{n^{\log_3(2)}}{n} \right) + c_0 n^{\log_3(2)} \\
= 3c_1 n - 3c_1 n^{\log_3(2)} + c_0 n^{\log_3(2)} \\
= 3c_1 n + (c_0 - 3c_1)n^{\log_3(2)}
\]

Since \( n^{\log_3(2)} \leq n^{\log_3(3)} = n \) can conclude that the \( 3c_1 n \) term dominates the asymptotic runtime, so the function is, indeed, \( O(n) \).

**MULTI-pop**

Consider augmenting the Stack ADT with an extra operation:

\texttt{multipop(k)}: Pops up to \( k \) elements from the Stack and returns the number of elements it popped

What is the amortized cost of a series of push’s, Stack assuming push and pop are both \( O(1) \)?

**Solution:**

Consider an empty Stack. If we run various operations (multipop, pop, and push) on the Stack until it is once again empty, we see the following: Note that \( \text{multipop}(k) \) takes \( ck \) time. If over the course of running the operations, we push \( n \) items, then each item is associated with at most one multipop or pop. It follows that the largest amount of time the multipops can take in aggregate is \( n \). Note that the smallest possible number of operations to amortize over is \( n + 1 \) (\( n \) pushes and 1 multipop). So, the worst amortized cost of a series of pushes, pops, and multipops is \( \frac{2n}{n + 1} = O(1) \). Where \( 2n \) comes from \( n \) pushes + \( n \) for the largest amount of time the multipops can take. The denominator comes from \( n \) pushes and 1 multipop (The smallest number of operations we could have that would take this long.).